CMPSC-265
Data Structures and Algorithms

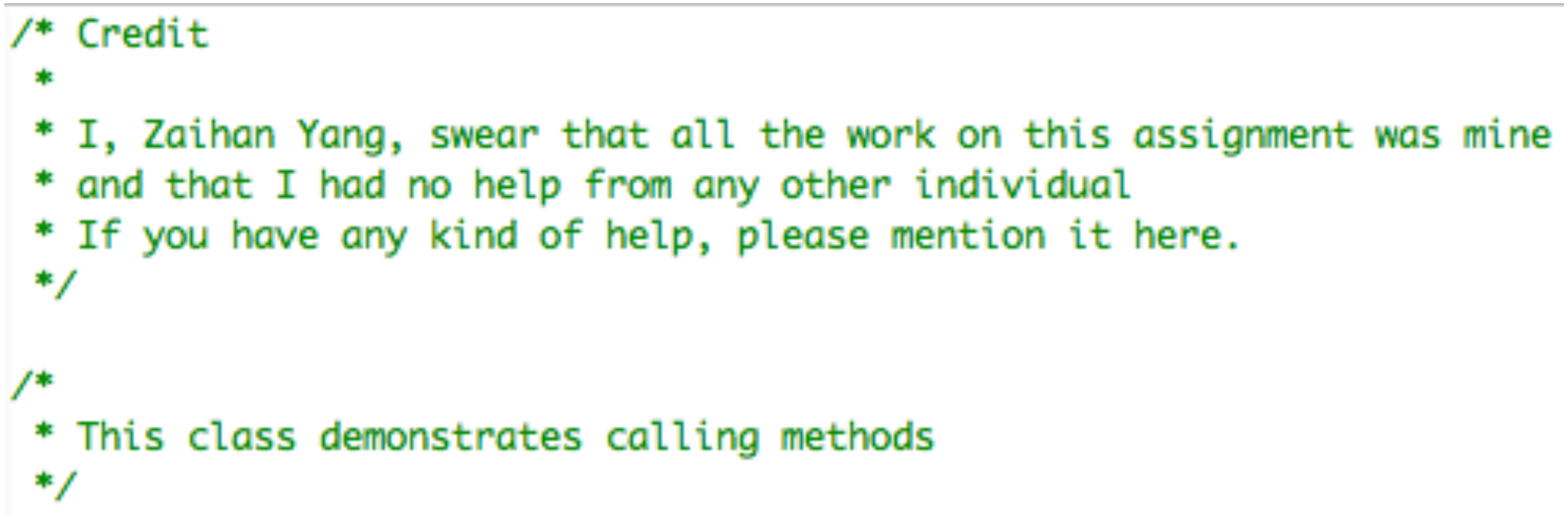
Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

- About <u>credit comments and comments</u>



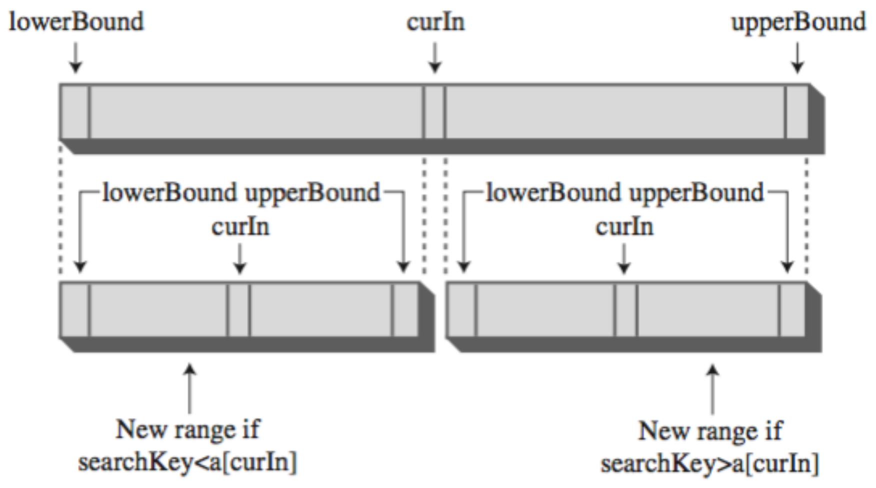
```
/* Credit
 *
 * I, Zaihan Yang, swear that all the work on this assignment was mine
 * and that I had no help from any other individual
 * If you have any kind of help, please mention it here.
 */

/*
 * This class demonstrates calling methods
 */
```

Recap

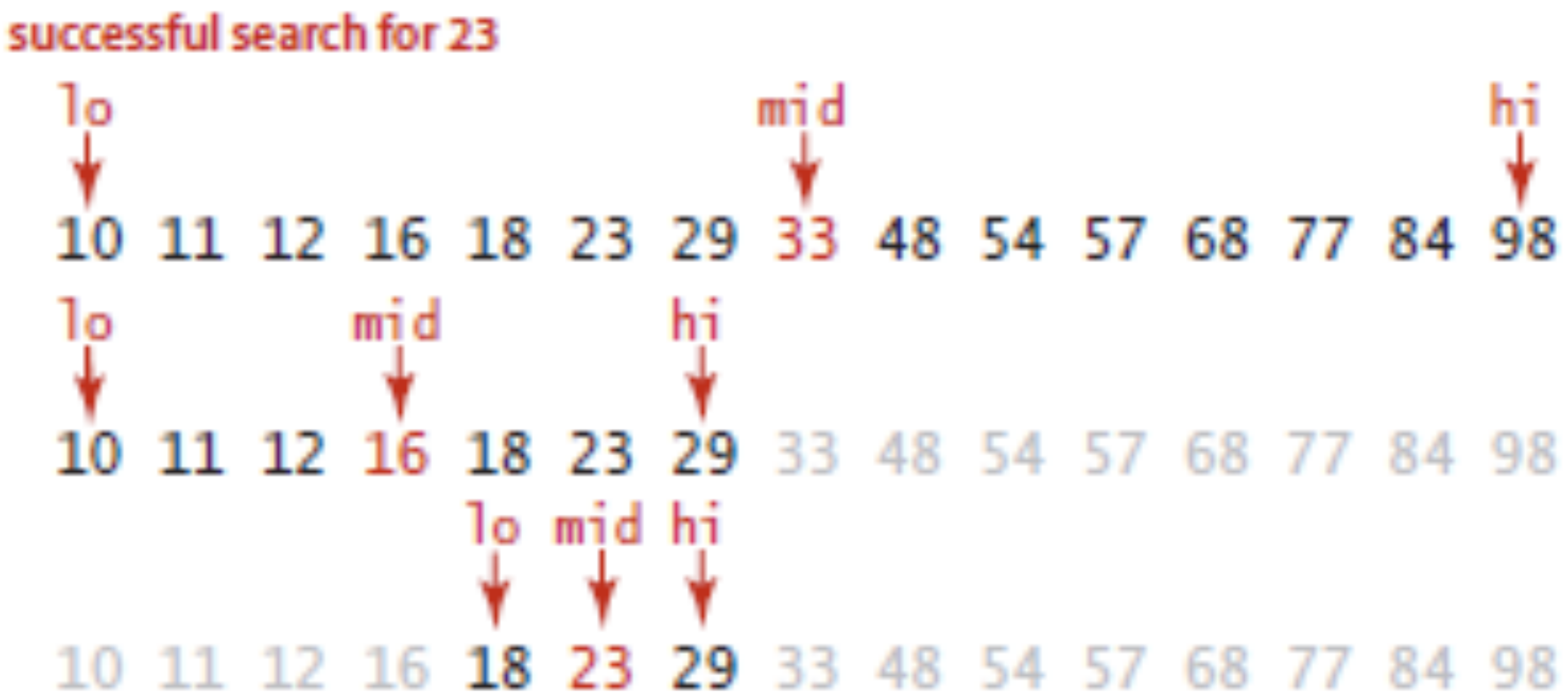
- Unordered array: HighArray.java
  - Insert
  - Delete
  - Swap
  - Search (linear search)
- Ordered array: OrdArray.java
  - Insert
  - Delete
  - Search (binary search)

Binary Search Implementation

```
public int find(int searchKey) {
int lowerBound = 0;
int upperBound = nElems-1;
int curIn;
while(true) {
  curIn = (lowerBound + upperBound) / 2;
  if(a[curIn]==searchKey)
      return curIn; // found it
else if(lowerBound > upperBound)
      return nElems; // not found
else // divide range {
  if(a[curIn] < searchKey)
     lowerBound = curIn + 1; //in upper half
 else
    upperBound = curIn - 1; //in lower half
} // end else divide range
} // end while
} // end find()
```



Binary Search

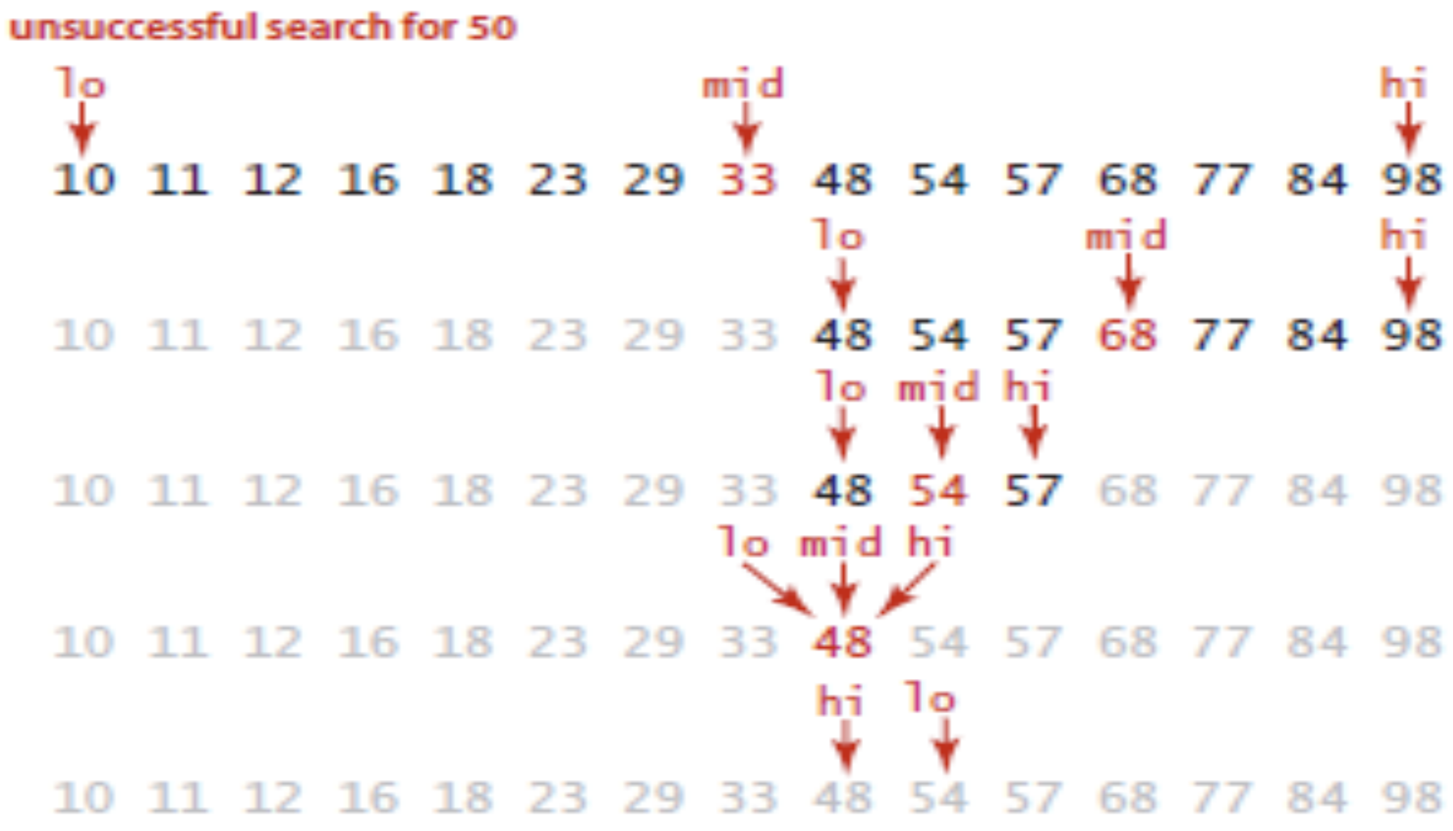


N=15.

3 iterations divide in half in each iteration

~ logN

Binary Search



N=15.
5 iterations divide in half in each iteration
~ logN

How Fast is Binary Search?

- Let’s find the number of steps needed to find the element
  - At each step, we eliminate half of the search space
  - We continue until one element is left (worst case)
  - Example, n=100 elements

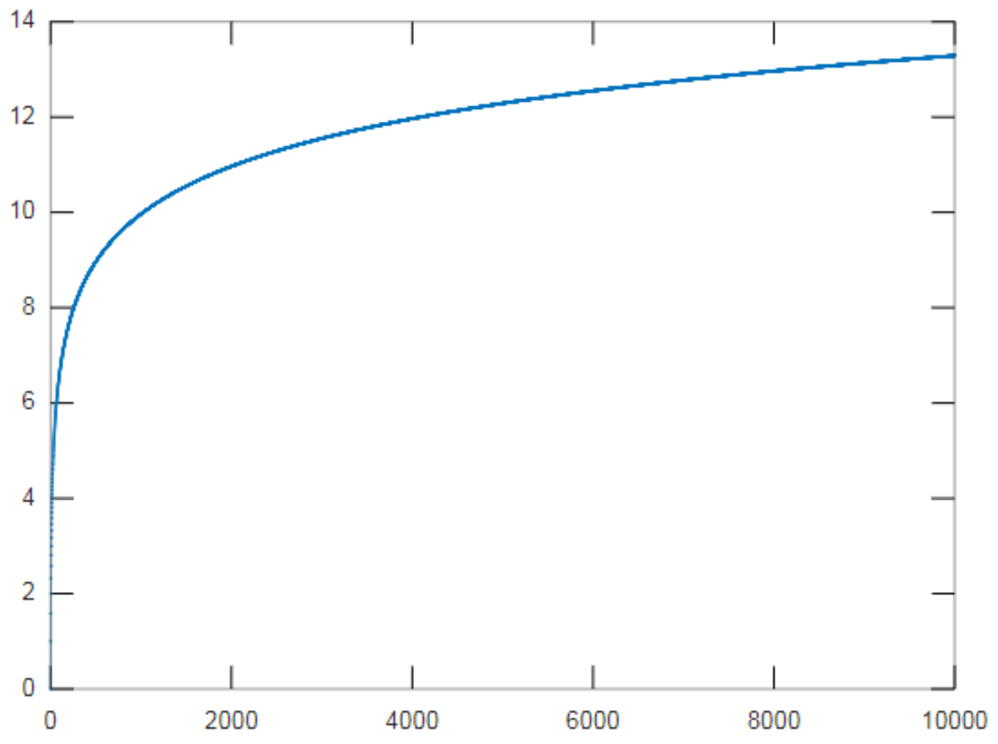
Step	n		Number of items
1	100/2 = 50.0000	$(n/2^1)$	50
2	50/2 = 25.0000	$(n/2^2)$	25
3	25/2 = 12.5000	$(n/2^3)$	13
4	12.5/2 = 6.25000	$(n/2^4)$	7
5	6.25/2 = 3.12500	$(n/2^5)$	4
6	3.125/2 = 1.56250	$(n/2^6)$	2
7	1.5625/2 = 0.78125	$(n/2^7)$	**1**

We at most need to guess **7** times to successfully find out the secret number.

How Fast is Binary Search?

- Let x be the number of steps needed to find the element in the worst case (having one element left).
  - $n / 2^x = 1$ or $n = 2^x$

$$\boxed{x = \log_2(n)}$$



Linear Search Vs Binary Search?

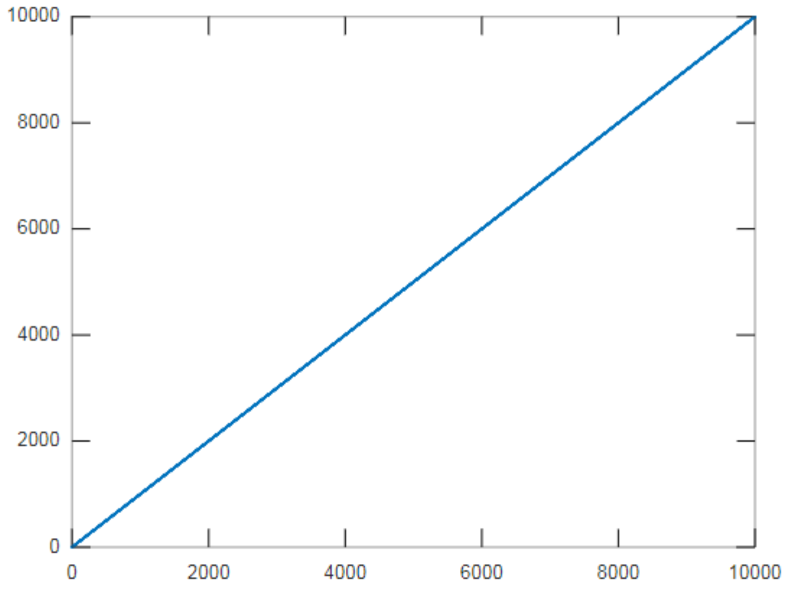
n	Linear search	Binary search
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24
100,000,000	100,000,000	27
1,000,000,000	1,000,000,000	**30**

Notice the difference between linear search and binary search. For very small numbers, the difference is not dramatic. However, the more items there are, the bigger the difference.

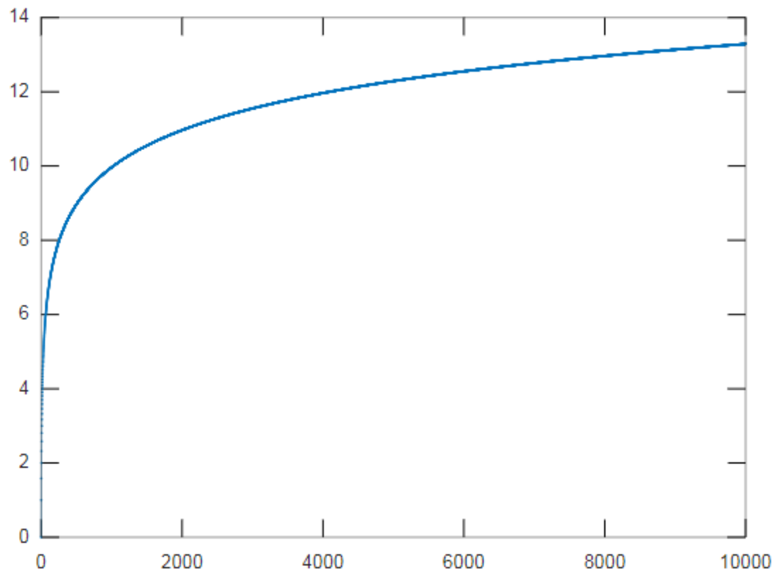
We say that for all but very small inputs, the binary search is greatly superior.

From Previous Session

- Binary search is more efficient than linear search



Linear growth



Logarithmic growth

Analysis of Algorithms

- What is efficiency?
  - Running time and memory needed.
- Programmers are interested in finding out:
  - How long will my program take?
    - It concerns about the **time efficiency** of a program.
  - How much memory does my program need to take?
    - It concerns about the memory (**space) efficiency** of a program.

How to measure efficiency

- People can carry out experiments to really calculate and keep track of the running time of the program. (Tic Toc)
- However:
  - Different computers, based upon their different architectures, CPUs and operating systems, might give different measures of time.
  - It is sometimes infeasible to implement all the algorithms, and run all the experiments with various input size.

Complexity Analysis

- People can also build *mathematical models* to analyze efficiency: Complexity Analysis
- We use time complexity to model time efficiency, and space complexity to mode space efficiency.
  - How the running time or required memory is related to the size of the data structure used by algorithms
- Trade-off based on application domain.
- Complexity analysis gives us a function that maps from input size to the number of steps the algorithm takes.
  - Since we want to compare algorithms, we are interested in a class of complexity, not the exact each individual function.

Examples

- Inserting into a general unordered array takes one step (add at the end of the array), Suppose it takes *c* seconds depending on the computer, compiler,…
  - We care about that *1* step not the exact time in seconds
    - We analyze the *algorithm* not the *implementation*
    - Constants can be eliminated
    - So we say that the complexity of this insertion operation is in order of 1.

- Linear search takes *n* steps, if each comparison takes *c* seconds the total time is c*n seconds
  - We say the complexity is in order of *n*

Big O Notation

- Defines the complexity order
  - Insert in array is in order of 1, so we denote it as O(1)
  - Linear search is in order n, so we denote it as O(n).

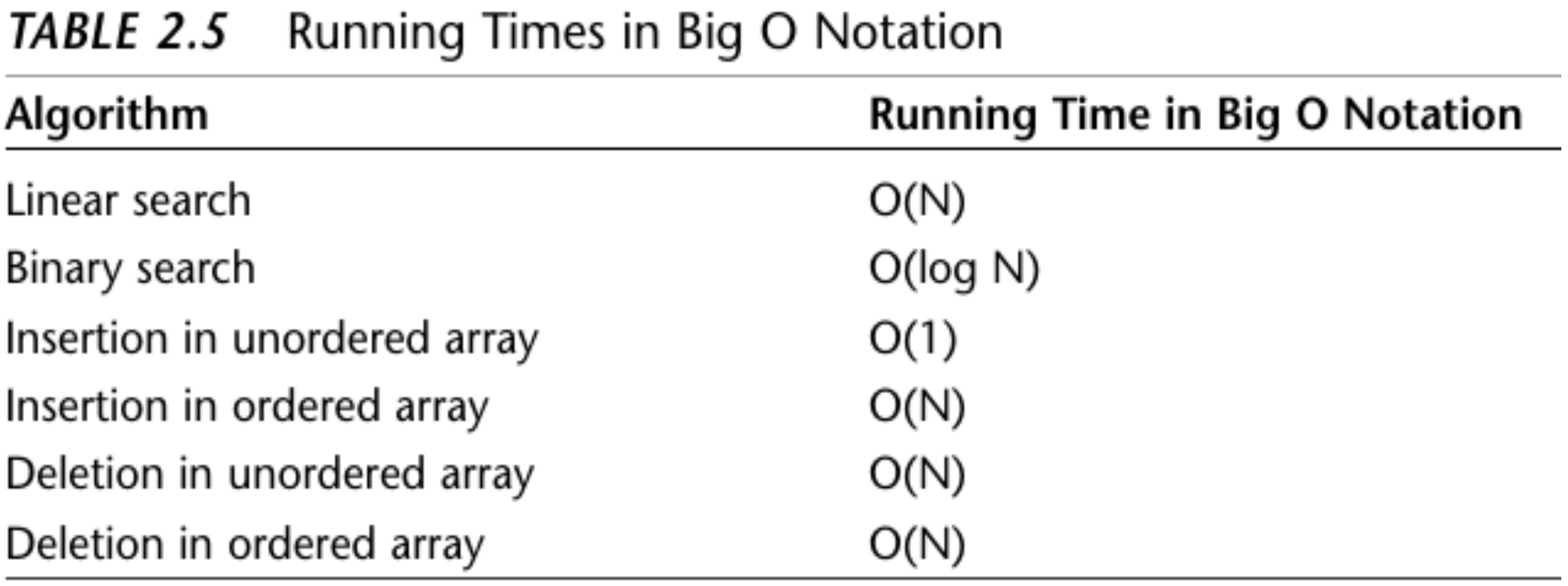


TABLE 2.5 Running Times in Big O Notation

Algorithm	**Running Time in Big O Notation**
Linear search	O(N)
Binary search	O(log N)
Insertion in unordered array	O(1)
Insertion in ordered array	O(N)
Deletion in unordered array	O(N)
Deletion in ordered array	O(N)

Big O Notation

- **Formal definition:**
- A function g(n) is O(f(n)) (read: g(n) is Big O of f(n)) if and only if there is a positive integer $n_0 \in N$ and a constant c > 0 such that for all $n \geq n_0$, we have that $g(n) \leq c * f(n)$.
  - g(n) is *eventually equal or smaller* than c ∗ f(n), which means g(n) will eventually grow slower than (or at most as fast as) f(n)
  - f(n) therefore determines the asymptotic upper-bounds of the growth of g(n)
  - We can write as g(n) = O(f(n)) or $g(n) \in O(f(n))$ indicating g(n) is a member of the set of O(f(n)) of functions, each of which are increasing with the same or less rat of f(n) when n->∞

Big O

Suppose g(n)=10,000 + 10n. Try prove g(n)=O(f(n)), where f(n)=n.

To proof: we need to find two positive numbers c and n0, so that for all n>=n0, g(n) will always be less than or equal to f(n)

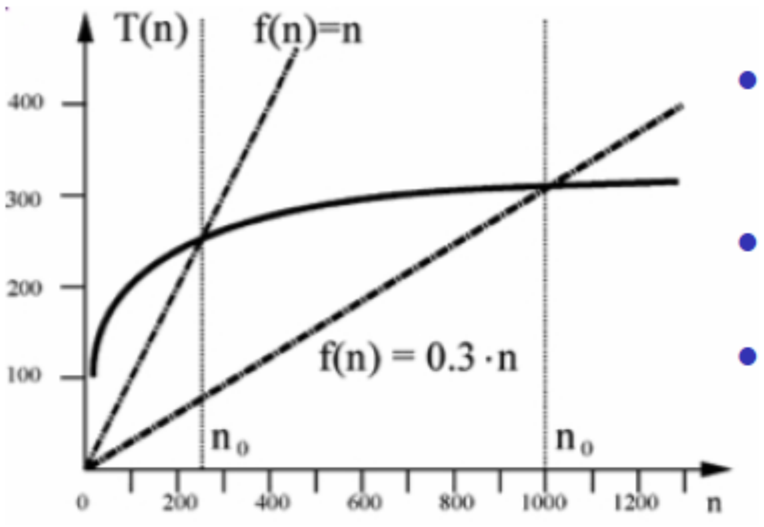
How about we choose c=20, and n0=1000.
G(n) andn f(n) will intersect at the point (1000, 20,000), and after that, g(n) is always less than f(n).
f(n) = 10,000+10n<20n (for n>=1000).
And therefore, we say, g(n)=O(f(n)) or $g(n) \in O(f(n))$

Big O

Example 1.8; p.13: $g(n) = 100\log_{10} n$ is $O(n)$

$$g(n) < n \text{ if } n > 238 \text{ or } g(n) < 0.3n \text{ if } n > 1000.$$

By definition, $g(n)$ is $O(f(n))$, or $g(n) = O(f(n))$ if a constant $c > 0$ exists, such that $cf(n)$ grows faster than $g(n)$ for all $n > n_0$.



- To prove that some $g(n)$ is $O(f(n))$ means to show that for g and f such constants c and n_0 exist.
- The constants c and n_0 are interdependent.
- $g(n)$ is $O(f(n))$ iff the graph of $g(n)$ is always below or at the graph of $cf(n)$ after n_0.

Big O

$g1(n)= n$
$g2(n)= 2n$
$g3(n)= n/2$
$g4(n)=n + n/2 + 5$
$g5(n)=100000n+100.$

As we said, O(n) is a set of functions. All the above functions are all members of O(n). We can say they are all big O of n.

Big O Notation

- How to determine the Big O notation of an algorithm (a program of codes)
  - Analyze and represent the number of steps as a function: g(n)
  - Discard all the lower-order terms in g(n)
    - O(f(n)) generalizes the asymptotic upper bound of g(n). When, the contribution of the lower-order terms can be neglected.
  - Ignore all constants
  - The resulting function is the f(n)
  - Can denote g(n) = O(f(n))

Examples

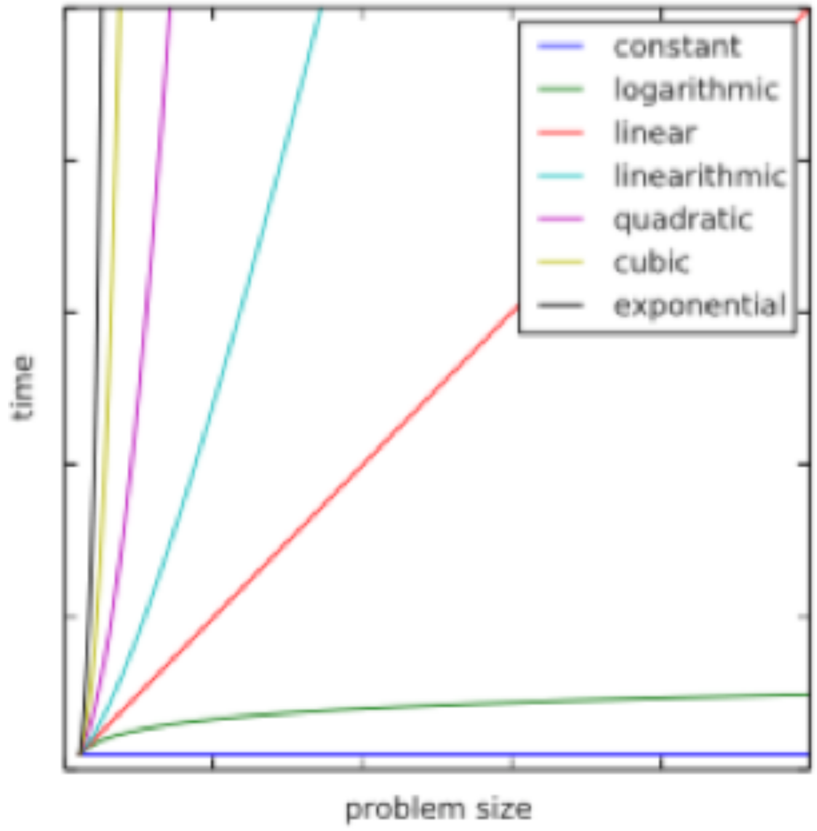
Number of steps (g(n))	Complexity order (Big O) f(n)
2*n + 100	
5*n + n^2	
50000*n + 0.0005*n^2	
100 + log(1000)	
n*(1+ log(n))	
$(n + 1)^2$	
$(n + \log(n))^2$	

Examples

Number of steps	Complexity order (Big O)
2*n + 100	O(n)
5*n + n^2	O(n^2)
50000*n + 0.0005*n^2	O(n^2)
100 + log(1000)	O(1)
n*(1+ log(n))	O(n*log(n))
$(n + 1)^2$	O(n^2)
$(n + \log(n))^2$	O(n^2)

Common Complexity Classes

- Some common complexity classes in order:
  - O(1), *constant*
  - O(log (n)), *logarithmic*
  - O(n), *linear*
  - O(n log(n)), *log-linear*
  - O(n^2), *quadratic*
  - O(n^3), *cubic*
  - O(2^n), *exponential*



  - *In terms of growth of rates:*
  - *$O(1)<O(\log n)<O(n)<O(n \log n)<O(n^2)<O(n^3)<O(2^n)$*

Growth Rates Example

Complexity	n = 4	n = 16	n = 256
O(1)	1	1	1
O(log(n))	2	4	8
O(n)	4	16	256
O(n*log(n))	8	64	2048
$O(n^2)$	16	256	65536
$O(n^3)$	64	4096	16777216
$O(2^n)$	16	65536	$1.16 * 10^{77}$

Big O Notation

- How to determine the Big O notation of an algorithm (a program of codes)
  - Analyze and represent the number of steps as a function: g(n)
  - How this can be done:
    - Develop an input model, including a definition of the problem size.
    - Identify the inner loop
    - Define a cost model that includes operations in the inner loop
    - Determine the frequency of execution of those operations for the given input.

Big O Notation: example

- What is the time complexity in terms of Big-O notation of the following segments of codes?

```
int sum=0;
for (int i=0; i<N; i++){
        sum++;
}
```

- The input size is: N
- One loop: N iterations
- The cost model is the number of times the statement "sum++" is done.
- Each iteration update sum once. So N iterations, the frequency is N

g(n)=n
Therefore: g(n)=O(n)

Big O Notation: example

```
int sum=0;
for (int i=1; i<=N; i++){
    for (int j=1; j<=N; j++){
        sum++;
    }
}
```

$g(n)=n^2$, and therefore $f(n)=n^2$
$g(n) = O(n^2)$

- The input size is: N
- Nested loop: N*N iterations
- The cost model is the number of times the statement "sum++" is done.
- Each iteration update sum once. So N*N iterations, the frequency is $N*N=N^2$

Big O Notation: example

```
int sum=0;
for (int i=1; i<=N; i*=2){
     for (int j=0; j<i; j++){
          sum++;
     }
}
```

1+2+4+8+16+….+N=2N-1~N.

Order of growth: N

Big O Notation: Example

```
int sum=0;
for (int i=0; i<N; i++) {
  for (int j=0; j<i*2; j++) {
    for (int k=0; k<10; k++){
      sum++;
    }
  }
}
```

Big O Notation: Example

```
int sum=0;
for (int i=0; i<N; i++) {
  for (int j=0; j<2*i; j++) {
    for (int k=0; k<10; k++){
      sum++;
    }
  }
}
```

Order of growth: N^2

Big O Notation: Example

```
int sum=0;
for (int i=0; i<N; i++) {
  for (int j=0; j<i*2; j++) {
    for (int k=0; k<10; k++){
      sum++;
    }
  }
}
```

Big O Notation: Example

```
int sum=0;
for (int i=0; i<N; i++) {
  for (int j=0; j<2*i; j++) {
    for (int k=0; k<10; k++){
      sum++;
    }
  }
}
```

Order of growth: N^2

Big O Notation: More Practice

```
for (int i=0;i<n;i++)
{
        for (int j=0;j<5;j++)
        {
                //some statement
        }
}
```

O(n)

```
for (int i=0;i<n;i++)
{
        for (int j=0;j<2*n;j++)
        {
                //some statement
        }
}
```

$O(n^2)$

Big O Notation: More Practice

```
for (int i=1;i<=n;i=i*2)
{
        //some statement
}
```

O(logn)

```
for (int i=0;i<n;i++)
{
        for (int j=i;j<n;j++)
        {
                //some statement
        }
}
```

O(n^2)

Big O Notation: More Practice

```
for (int i=1;i<n;i++)
{
        for (int j=0;j<m;j++)
        {
                //some statement
        }
}
```

O(nm)

```
for (int i=1;i<n;i++)
{
        for (int j=0;j<m;j++)
        {
                for (int k=0;k<5;k++)
                {
                        //some statement
                }
        }
}
```

O(nm)

In-Class Quiz

- Inserting an item into unordered array:
  - A) takes time proportional to the size of the array
  - B) requires multiple comparisons
  - C) requires shifting other items to make room.
  - D) takes the same time no matter how many items there are.
- True or False:
  - <u>In an unordered array, it is generally faster to find out an item is not in the array than to find out it is.</u>
- Ordered arrays, compared with unordered arrays, are
  - Much quicker at deletion
  - Quicker at insertion
  - Slower at insertion
  - Quicker at searching

In-Class Quiz

- The maximum number of elements that must be examined to complete a binary search in an array of 200 elements is:
    - A) 200
    - B) 8
    - C) 1
    - D) 13
- O(1) means a process operates in _____ time

In-Class Quiz

- Practice on Big-O notation
- Ex1.
  - Given g(n)=0.5n^3+5*n^2+2*n-1, and f(n)=n^3. Please prove g(n)=O(f(n)).